Workshop

Day 3:
The binary GA

Functional Programming and Intelligent Algorithms:
Genetic Algorithms
Spring 2015
Faculty of Engineering and Natural Sciences
Ålesund University College

Robin T. Bye*

Revision date: 19 February 2015

*email: roby@hials.no; website: www.robinbye.com; phone: 70 16 15 49; office: B314.

Contents

1 Workshop overview

1.1 Topics

Today's topics include:

- Components of the binary GA
- Solving a 1D and 2D functional problems using a binary GA
- String-learning GA

1.2 Reading material

Compulsory reading to be studied *before* this workshop is Chapter 2 in Haupt & Haupt (2004) on binary GAs.

1.3 Specific learning outcomes

After completing this workshop, including self-study, reading and exercises, the students should be able to

- explain the components and algorithmic flow of the binary GA.
- demonstrate typical effects of changing parameters of the binary GA.

- implement and modify their own binary GAs to suit a variety of problems, including function optimisation and toy problems such as string learning.

1.4 Schedule

We begin at 8.15 with a recap of the last couple of weeks' activities and questions. Today's workshop will roughly follow the schedule below:

08.15 Status update/recap.

08.45 Binary GA.

10.15 Workshop rest of the day.

Description of a Basic GA

The basic steps that almost any GA consists of are outlined in high-level pseudocode in Algorithm 1 below, where we adopt a cost function as our objective function (loosely adapted from Haupt & Haupt, 2004, and other sources). Note that the pseudocode in Algorithm 1 is procedural, whereas you need to reformulate it to being functional in Haskell (of course, since you will be doing input/output and randomness, some impurity is unavoidable).

A chromosome c is an encoded candidate solution to the optimisation problem of optimising $f(c)$. The design parameters that we want to optimise must be translated (encoded) from their original domain to a format suitable for the GA, usually arrays of bits or real-valued numbers, often normalised to the interval $[0, 1]$. The bits or numbers are usually called genes.

The objective function quantifies that quality of candidate solutions, that is, how well they fulfill the desired design criteria. The selection criterion determines how many chromosomes in a population survives from one iteration to the next. For examle, using the roulette wheel method, the cost (fitness) associated with each chromosome is evaluated and the chromosomes are given a weighted selection probability according to their cost, where a smaller cost (greater fitness) results in a greater probability.

A pre-determined fraction, of chromosomes (typically half the population) is then randomly picked, with low cost (high fitness) chromosomes having a greater chance of being picked and kept for survival and reproduction.

For mating, several crossover methods exists, where genes from two parent chromosomes are combined into one or several offspring, which are then put back into the population, replacing those chromosomes that were not selected for mating.

After mating, a fraction of the chromosomes will have one or several of their genes mutated. This means flipping (inverting) bits for binary chromosomes, or changing the values of these genes to


```
/* INITIALISATION                                                    */
define encoding scheme for chromosomes c;
define objective function f(c);
set criteria for selection, crossover, mutation, elitism;
generate initial population of chromosomes;
sort population in increasing order of cost;
bestChrom ← population[0];
set minCost, maxIterations;
i ← 1;
/* LOOP                                                              */
while i < maxIterations OR bestCost > minCost do
    evaluate cost for each chromosome;
    select chromosomes for mating;
    perform mating, crossover, mutation, elitism;
    update population;
    sort population in increasing order of cost;
    bestChrom ← population[0];
    bestCost ← f(bestChrom);
    i ← i + 1;
end
return i, bestChrom, bestCost;
decode bestChrom to original domain;
```

Algorithm 1: Basic GA.

random numbers within some allowable range.

Next, each of the chromosomes in the updated population is evaluated by the objective function and the population is sorted in descending order of performance (ability to minimise cost or maximise fitness).

The process repeats until the maximum number of iterations has been reached, or the solution (the best chromosome) has reached a satisfactory performance. Then the algorithm ends and returns the best chromosome, which is decoded back to its original domain. In our case, the decoded solution specifies the optimal values for selected design parameters of an offshore crane.

2 Exercises

2.1 Components of the binary GA

Exercise 2.1: Draw a diagram depicting the steps that constitute the algorithmic flow of a binary GA.

Exercise 2.2: Explain the following terms:

(a) N_{var}

(b) N_{gene}

(c) N_{bits}

(d) N_{pop}

(e) X_{rate}

(f) N_{keep}

Exercise 2.3: In terms of search and optimisation algorithms such as the GA, what is meant by the following terms:

(a) Elitism.

(b) Convergence rate.

(c) Exploration.

(d) Exploitation.

Exercise 2.4: Suppose you have a GA with some parameter settings that yield a particular level of exploration and a particular rate of convergence (exploitation). Explain what is likely to happen if you increase or decrease

(a) the population size.

(b) the mutation rate.

(c) the selection rate.

(You may have to come back to this question after having tried it for yourself in the following exercises).

2.2 String learning using a binary GA

Implement a GA able to learn a string. You should use characters from the entire English alphabet consisting of 26 characters from 'a' to 'z'; ten digits from '0' to '9'; and the following twelve symbols:
`. , ; : ? ! _ + - * / ␣`
That is, your character set will have a total of 48 characters. Note that the symbol ␣ means the space character. For convience, here is the entire alphabet:

```
alphabet = "abcdefghijklmnopqrstuvxyz0123456789.,;:?!_+-*/ "
```


You should create a module called BinaryGA.hs where you will enter all your code.

Exercise 2.5: How many bits are needed to represent a character? Suggest an encoding scheme and implement it in a function called `encodeString`.

Exercise 2.6: Implement a decoding function `decodeChromosome` that decodes a chromosome (bit string or list of bits) and returns the corresponding character string. Test and demonstrate that your function works by supplying test chromosomes and observe if the function generates correct character strings.

Exercise 2.7: Test and demonstrate that your encoding and decoding functions work by encoding test strings of characters and then decoding the encoded strings to see if you get back the original strings:

```
s5 = "descartes: cogito ergo sum"
b = encodeString s5
s' = decodeChromosome b
itWorks = s5 == s'
```

Exercise 2.8: Implement a cost function `stringCost` with two arguments `s` (test string) and `c` (correct string). The function should return a cost based on the difference between the test string `s` and the correct string `c`. Test the function by supplying some test strings to it:

```
s1 = "abcde"
s2 = "aaaaa"
S3 = "hxlxo"
c = "hello"
cost1 = stringCost s1 c
cost2 = stringCost s2 c
cost3 = stringCost s3 c
```

Note that we could have calculated a cost based on chromosomes instead. However, for most problems, the cost function will relate to real-world values (e.g., the x and y coordinates of the 2D optimisation problem from Haupt & Haupt (2004) used as an example in the lectures).

Exercise 2.9: Implement a binary GA `stringLearningGA`. with one argument `s` (character string). The algorithm should be able to learn the character string `s`.

NOTE: More code suggestions will come here later but please start yourself!

Some things you will need to consider are how to implement a GA such as the one in Algorithm 1, that is,

(a) a selection method (from top to bottom, random, cost weighted, rank weighted, tournament selection)

(b) a crossover method (single-point, double-point, uniform, etc.)

(c) a mutation method

(d) an elitist method

(e) a method for sorting the population in increasing order of cost

(f) a method for updating the population between generations

(g) randomness (which is impure)

Exercise 2.10: Test your binary GA on some test strings such as

- `abc123.,;`
- `descartes: cogito ergo sum`
- `2+2 is: 4, 2*2 is: 4; why is _/not/_ 2.2*2.2 equal to 4.4?!`

Experiment with algorithm settings such as

- population size (number of chromosomes)
- maximum number of iterations
- selection rate
- mutation rate

and discuss your results in terms of ability to find the correct string, e.g., examine the

- GA-generated string vs. correct string
- cost of GA-generated string (zero for finding the correct string)
- number of iterations needed
- total runtime

If you modify your cost function `stringCost`, or implement other cost functions, examine the performance of the GA for each of these.

2.3 Function optimisation using a binary GA

Exercise 2.11: Modify your binary GA from above or write a new one to optimise the test functions given in Appendix I of Haupt & Haupt (2004) (you have already implemented f_1, f_2, f_6, and f_7 previously). Does it work? Experiment with different algorithm settings, such as varying the

- population size (number of chromosomes)
- maximum number of iterations

- selection rate
- mutation rate

How does your algorithm compare with your simple minimum-search algorithm `fminsimple` used previously?

Also, note that the test function f_7 got an erroneous solution in Haupt & Haupt (2004). The correct solution is

$$x_m = 9.039 \tag{2.1}$$
$$y_m = 8.668 \tag{2.2}$$
$$f_{min}(x_m, y_m) = -18.5547 \tag{2.3}$$

That is, both x_m and y_m are wrong by a factor of 10.

3 Homework

- Complete all the exercises above.
- Read through (again!) the specific learning outcomes in Section 1.3 to check which outcomes you have not attained yet. Study today's material and prepare questions for tomorrow about learning outcomes you have missed.
- Prepare for Day 4 by reading about the continuous genetic algorithm (GA) in Chapter 3 of Haupt & Haupt (2004).

References

Haupt, R. L., & Haupt, S. E. (2004). *Practical Genetic Algorithms*. Wiley, 2nd ed.

